

**IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
BEFORE THE BOARD OF PATENT APPEALS AND INTERFERENCES**

In re Application of:)	
)	
YODAIKEN ET AL.)	
)	
Serial No.: 10/670,802)	Group Art Unit: 2191
)	
Filed: September 26, 2003)	Examiner: Ted T. Vo
)	
For: SYSTEM AND METHOD FOR)	Board of Patent Appeals and
DYNAMICALLY LINKING)	Interferences
APPLICATIONS SOFTWARE INTO)	
A RUNNING OPERATING SYSTEM)	
KERNEL)	
)	
Confirmation No.: 2648)	

Mail Stop: Appeal Brief – Patents
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

APPEAL BRIEF UNDER 37 C.F.R. § 41.37

In support of the Notice of Appeal filed on August 11, 2009, and pursuant to 37 C.F.R. § 41.37, Appellants present this Appeal Brief in the above-captioned application.

This is an appeal to the Board of Patent Appeals and Interferences from the Examiner's final rejection of claims 1, 2, 4-7, and 10-68 in the Final Office Action dated April 28, 2009. The appealed claims are set forth in the attached Claims Appendix.

1. Real Party in Interest

This application is assigned to Wind River Systems, Inc., the real party in interest.

2. Related Appeals and Interferences

There are no other appeals or interferences that would directly affect, be directly affected, or have a bearing on the instant appeal.

3. Status of the Claims

Claims 1, 2, 4-7, and 10-68 have been rejected in the April 28, 2009 Final Office Action. Claims 3, 8, and 9 were canceled in a previous amendment. The final rejection of claims 1, 2, 4-7, and 10-68 is being appealed.

4. Status of Amendments

All amendments submitted by Appellants have been entered.

5. Summary of Claimed Subject Matter

The present invention, as recited in independent claim 1, relates to a system for dynamically linking application software into a running operating system kernel. (See Specification, p. 4, ¶ [008]). The system comprises an environment library comprising one or more routines for insulating the application code from the operating system environment and for implementing a uniform execution environment, wherein the insulating of the application code allows the application code to perform a function by executing a routine that is standard to the application code instead of by using a routine that pertains to the running operating system kernel. (See Id., pp. 5-6, ¶ [0010]; p. 9, ¶ [0023]; and pp. 13-14, ¶¶ [0035]-[0036]). The system further comprises a build system for constructing a loadable module from the application code and the environment library and for constructing a standard executable program from the

loadable module and an execution library. (See Id., pp. 9-10, ¶¶ [0024]-[0025]; p. 17, ¶ [0046]); and Figs. 2-4). The loadable module including a module input and a module output, and wherein the execution library comprises one or more routines for transparently loading the loadable module into the running operating system kernel, passing arguments to the loadable module, and terminating and unloading the loadable module after receiving a termination signal. (See Id., pp. 5-6, ¶ [0010]; pp. 10-11, ¶¶ [0026]-[0028]; and p. 12, ¶ [0031]). In addition, the one or more routines of the execution library setting up input/output channels by connecting a standard input and a standard output of the running operating system kernel to the module input and the module output. (See Id., p. 10, ¶ [0026]; and pp. 12-13, ¶ [0033]).

The present invention, as recited in independent claim 5, is directed to a method for dynamically linking application software into a running operating system kernel. (See Id., p. 4, ¶ [008]; and pp. 5-6, ¶¶ [0010]-[0011]). The method comprises creating a loadable module, the loadable module including a module input and a module output, creating an executable program, and executing the executable program. (See Id., pp. 9-10, ¶¶ [0024]-[0025]; p. 17, ¶ [0046]; and Figs. 2-4). The executable program performs a method comprising the step of setting up input/output channels by connecting a standard input and a standard output of a running operating system kernel to the module input and the module output. (See Id., pp. 5-6, ¶ [0010]; pp. 10-11, ¶¶ [0026]-[0028]; and p. 12, ¶ [0031]). The method performed by the executable program further comprises the step of inserting the loadable module into address space of the running operating system kernel, wherein, once the loadable module is inserted into the address space, the loadable module begins to execute. (See Id., p. 10, ¶ [0026]; and pp. 13-14, ¶ [0036]). The method performed by the executable program further comprises the step of waiting for the loadable module to connect via kernel/user channels and then connecting those kernel/user channels to the input/output channels, wherein the executable program is insulated from the running operating system kernel. (See Id., pp. 10-11, ¶ [0027]-[0028]; pp. 12-13, ¶ [0033]; and pp. 15-16, ¶¶ [0039]-[0040]). The insulating of the executable program allows the executable program to perform a function by executing a routine that is standard to the executable program instead of by using a routine that pertains to the running operating system kernel. (See Id., pp. 5-6, ¶ [0010]; and pp. 13-14, ¶¶ [0035]-[0036]).

The present invention, as recited in independent claim 21, is directed to a computer readable storage medium provided with a set of instructions executable by a processor for dynamically linking application software into a running operating system kernel. (See Id., p. 4, ¶ [008] ; and pp. 5-6, ¶¶ [0010]-[0011]). The set of instructions operable to a first set of computer instructions for insulating application code from an operating system environment, wherein the insulating of the application code allows the application code to perform a function by executing a routine that is standard to the application code instead of by using a routine that pertains to the running operating system kernel. (See Id., pp. 5-6, ¶ [0010]; and pp. 13-14, ¶¶ [0035]-[0036]). The set of instructions further operable to a second set of computer instructions for constructing a loadable module from the application code and the first set of computer instructions, the loadable module including a module input and a module output. (See Id., pp. 9-10, ¶¶ [0024]-[0025]; p. 17, ¶ [0046]; and Figs. 2-4). The set of instructions operable to a third set of computer instructions for constructing an executable program from the loadable module. (See Id., pp. 5-6, ¶ [0010]; and p. 9, ¶ [0024]). The fourth set of computer instructions includes computer instructions for transparently loading the loadable module into a running operating system kernel, passing arguments to the loadable module, and terminating and unloading the loadable module from the running operating system kernel after receiving a termination signal. (See Id., pp. 5-6, ¶ [0010]; p. 8, ¶ [0022]; p. 19, ¶ [0052]; and pp. 20-21, ¶ [0057]). The fourth set further includes at least one routine setting up input/output channels by connecting a standard input and a standard output of the running operating system kernel to the module input and the module output. (See Id., pp. 5-6, ¶ [0010]; pp. 10-11, ¶¶ [0026]-[0028]; and p. 12, ¶ [0031]).

The present invention, as recited in independent claim 34, is directed to a computer system for dynamically linking application software into a running operating system kernel. (See Id., p. 4, ¶ [008] ; and pp. 5-6, ¶¶ [0010]-[0011]). The computer system comprises first means for insulating application code from an operating system environment, wherein the insulating of the application code allows the application code to perform a function by executing a routine that is standard to the application code instead of by using a routine that pertains to the running operating system kernel. (See Id., pp. 5-6, ¶ [0010]; and pp. 13-14, ¶¶ [0035]-[0036]). The computer system further comprises a second means for constructing a loadable module from the application code and the first means, the loadable module including a module input and a

module output. (*See Id.*, pp. 9-10, ¶¶ [0024]-[0025]; p. 17, ¶ [0046]; and Figs. 2-4). The computer system further comprises a third means for constructing an executable program from the loadable module. (*See Id.*, pp. 5-6, ¶ [0010]; and p. 9, ¶ [0024]). The computer system further comprises a fourth means for transparently loading the loadable module into a running operating system kernel, passing arguments to the loadable module, and terminating and unloading the loadable module from the running operating system kernel after receiving a termination signal. (*See Id.*, pp. 5-6, ¶ [0010]; p. 8, ¶ [0022]; p. 19, ¶ [0052]; and pp. 20-21, ¶ [0057]). The fourth means includes at least one routine setting up input/output channels by connecting a standard input and a standard output of the running operating system kernel to the module input and the module output. (*See Id.*, pp. 5-6, ¶ [0010]; pp. 10-11, ¶¶ [0026]-[0028]; and p. 12, ¶ [0031]).

The present invention, as recited in independent claim 47, is directed to a computer system for dynamically linking application code created by a programmer into a running operating system kernel. (*See Id.*, p. 4, ¶ [008] ; and pp. 5-6, ¶¶ [0010]-[0011]). The computer system comprises a means for creating a loadable module, the loadable module including a module input and a module output. (*See Id.*, pp. 9-10, ¶¶ [0024]-[0025]; p. 17, ¶ [0046]; and Figs. 2-4). The computer system further comprises a means for creating an executable program that is configured to perform a method. The method comprises the step of setting up input/output channels by connecting a standard input and a standard output of the running operating system kernel to the module input and the module output. (*See Id.*, pp. 5-6, ¶ [0010]; pp. 10-11, ¶¶ [0026]-[0028]; and p. 12, ¶ [0031]). The method further comprises the step of inserting the loadable module into address space of the running operating system kernel, wherein, once the loadable module is inserted into the address space, the loadable module begins to execute. (*See Id.*, p. 10, ¶ [0026]; and pp. 13-14, ¶ [0036]). The method further comprises the step of waiting for the loadable module to connect via kernel/user channels and then connecting those kernel/user channels to the input/output channels. (*See Id.*, pp. 10-11, ¶ [0027]-[0028]; pp. 12-13, ¶ [0033]; and pp. 15-16, ¶¶ [0039]-[0040]). The executable program is insulated from the running operating system kernel. (*See Id.*, pp. 13-14, ¶¶ [0035]-[0036]). The insulating of the executable program allows the executable program to perform a function by executing a routine that is standard to the executable program instead of by using a routine that

pertains to the running operating system kernel. (See Id., pp. 5-6, ¶ [0010]; and pp. 13-14, ¶¶ [0035]-[0036]).

The present invention, as recited in independent claim 52, is directed to a method for dynamically linking application code created by a user into a running operating system kernel. (See Id., p. 4, ¶ [008]). The method comprises constructing a loadable module from application source code written by a user, the loadable module including a module input and a module output. (See Id., pp. 9-10, ¶¶ [0024]-[0025]; p. 17, ¶ [0046]; and Figs. 2-4). The method further comprises setting up input/output channels by connecting a standard input and a standard output of the running operating system kernel to the module input and the module output. (See Id., pp. 5-6, ¶ [0010]; pp. 10-11, ¶¶ [0026]-[0028]; and p. 12, ¶ [0031]). The method further comprises creating an executable program, wherein the executable program is configured to transparently load the loadable module into the running operating system kernel, thereby loading the loadable module into the running operating system kernel. The method further comprises unloading the loadable module from the running operating system kernel by sending a termination signal to the executable program. (See Id., pp. 5-6, ¶ [0010]; p. 8, ¶ [0022]; p. 19, ¶ [0052]; and pp. 20-21, ¶ [0057]). The executable program is insulated from the running operating system kernel. (See Id., pp. 13-14, ¶¶ [0035]-[0036]). The insulating of the executable program allows the executable program to perform a function by executing a routine that is standard to the executable program instead of by using a routine that pertains to the running operating system kernel. (See Id., pp. 5-6, ¶ [0010]; and pp. 13-14, ¶¶ [0035]-[0036]).

6. Ground of Rejection to be Reviewed on Appeal

- I. Whether claims 34-51 are indefinite under 35 U.S.C. § 112, 2nd Paragraph, for failing to particularly point out and distinctly claim the subject matter.
- II. Whether claims 34-51 fail to meet 35 U.S.C. § 101, per se, for covering a “computer system”.

- III. Whether claims 1, 2, 4-7, and 10-68 are unpatentable under 35 U.S.C. § 103(a) over U.S. Patent No. 5,136,709 to Shirakabe et al. (hereinafter “Shirakabe”) in view of U.S. Patent No. 5,778,226 to Adams et al. (hereinafter “Adams”).

7. Argument

- I. Claims 34-51 should not be rejected under 35 U.S.C. § 112, 2nd Paragraph, for failing to particularly point out and distinctly claim the subject matter.

Claims 34-51 stand rejected under 35 U.S.C. § 112 as being indefinite for failing to particularly point out and distinctly claim the subject matter which the applicant regards as his invention.

According to the 04/28/2009 Final Office Action, the Examiner appears to assert in this rejection that a claim to a “computer system” must recite in its body elements in which the Examiner believes are indispensable to any “computer system” claim. Thus, the Examiner faults the claims for omitting such stock computer hardware terms as “processor,” “memory,” “peripheral devices,” etc. As noted in both the 02/12/2009 Amendment and the 06/29/2009 Response, Appellants had been unaware that the Patent Office imposes on computer system claims such a stringent requirement. In fact, in both the above-mentioned Amendment and Response, Appellants had respectfully requested for the Examiner to provide a specific citation to an unambiguous authority; whether it is in the form of a statute, rule, or precedent that deems “computer system” claims indefinite unless they recite these familiar terms. However, in the 08/05/2009 Advisory Action, the Examiner fails to provide such authority. Instead, the Examiner merely stated that Appellants’ argument have been repeated within the above-mentioned Amendment and Response. Accordingly the Examiner has yet to address the fact that no authority has been provided by the Examiner to support this rejection.

To reiterate the Examiner's reasoning for the 35 U.S.C. § 112, 2nd Paragraph, the Examiner states in the Response to Argument of the 04/28/2009 Final Office Action, without providing any basis or reference to an authority:

If the claims recite "computer system" as a narrow limitation, it falls in the broad range of means plus functions. The claims recite the means plus function without hardware connected to scope of "computer system", then the scope of the claim is indefinite. Furthermore, the means plus functions recited in the claims fail to be identified with the structure, material, or acts described in the specification as corresponding to each claimed functions. Within the scope of the claims, it is unable to interpret the term "computer system" as a hardware machine. Since its means plus functions cover all other things include [sic] software elements, the recitations render the claims indefinite. (See 04/28/2009 Final Office Action, p. 3, lines 1-8).

Appellants contend that without any citations to relevant laws, rules, cases, or precedent of any form, the arguments presented by the Examiner lack basis and are conclusionary, at best. While the Examiner may feel compelled to serve as his own authority when determining the patentability of a claim, Appellants would like to direct the Examiner's attention to a relevant portion of the Manual of Patent Examining Procedure, or "M.P.E.P." According to § 2181 of the M.P.E.P., entitled "Identifying a 35 U.S.C. 112, Sixth Paragraph Limitation":

This section sets forth guidelines for the examination of § 35 U.S.C. 112, sixth paragraph, "**means or step plus function**" limitations in a claim. These guidelines are based on the Office's current understanding of the law and are believed to be fully consistent with binding precedent of the Supreme Court, the Federal Circuit and the Federal Circuit's predecessor courts. These guidelines do not constitute substantive rulemaking and hence do not have the force and effect of law...

If one skilled in the art would be able to identify the structure, material or acts from the description in the specification for performing the recited function, then the requirements of 35 U.S.C. § 112, second paragraph, are satisfied. See Dossel, 115 F.3d at 946-47, 42 USPQ2d at 1885 (The function recited in the means-plus-function limitation involved "reconstructing" data. The issue was whether the structure underlying this "reconstructing" function was adequately described in the written description to satisfy 35 U.S.C. § 112, second paragraph. The court stated that "[n]either the written description nor the claims uses the magic word 'computer,' nor do they quote computer code that may be used in the invention. Nevertheless, when the written description is combined with claims 8 and 9, the disclosure satisfies the requirements of Section § 112, Para. 2." The court concluded that based on the specific facts of the case, one skilled in

the art would recognize the structure for performing the “reconstructing” function since “a unit which receives digital data, performs complex mathematical computations and outputs the results to a display must be implemented by or on a general or special purpose computer.”). See also Intel Corp. v. VIA Technologies, Inc, 319 F.3d 1357, 1366, 65 USPQ2d 1934, 1941 (Fed. Cir. 2003) (The “core logic” structure that was modified to perform a particular program was held to be adequate corresponding structure for a claimed function although the specification did not disclose internal circuitry of the core logic to show exactly how it must be modified.). (See M.P.E.P. § 2181, Introduction and (III) (B) (1)). (Emphasis added).

As noted by the Examiner, the specification clearly provides sufficient support for the claimed “computer system” performing the claimed “means plus function”, and even includes numerous references to the “magic word ‘computer.’” Specifically, the specification states: “In the following description, for purposes of explanation and not limitation, specific details are set forth, such as *particular systems, computers, devices, components, techniques, computer languages, storage techniques, software products and systems, operating systems, interfaces, hardware, etc. in order to provide a thorough understanding of the present invention...*” (See Specification, p. 2, ¶ [0023]). Furthermore, Fig. 7 provides a detailed illustration of an exemplary computer system, complete with processing units, memory, workstations (e.g., laptops, desktops, etc.), servers, interface devices, etc. (See Id., p. 5, ¶ [0059]; and Fig. 7). The Examiner goes so far as to state, “[i]t is known that a computer comprises the hardware elements such as processor, memory, peripheral devices, etc.” (See 04/28/2009 Final Office Action, p. 4, lines, 17-19). Accordingly, Appellants find it extremely difficult to believe that one of skill in the art of computer science would fail to understand the phrase “computer system” upon reading the claims, reading the specification, and viewing the figures of the present application. As noted above by the Federal Court, “[i]f one skilled in the art would be able to identify the structure, material or acts from the description in the specification for performing the recited function, then the requirements of 35 U.S.C. § 112, second paragraph, are satisfied.”

In reference to claim 34, these “means plus function” elements include means for “insulating application code...”, means for “constructing a loadable module...”, means for “constructing an executable program...”, and means for “loading the loadable module, passing

arguments, terminating and unloading the loadable module, etc.” Each of these means are performed by the claimed computer system, which one skilled in the art of computer science would clearly recognize as a structural element disclosed in the specification, namely a computer or other processing devices. Therefore, in contrast to the Examiner’s unsupported assertion that a claimed “computer system” is indefinite, the Specification clearly states “computer” and, the M.P.E.P. notes that one skilled in the art would recognize this computer as the structure for performing the “insulating application code,” constructing a loadable module,” “constructing an executable program,” “loading the loadable module,” etc. Thus, Appellants respectfully maintain the position that the language in claim 34 is definite and the claim should not have been rejected under 35 U.S.C. § 112. Because claims 35-46 depend from, and, therefore, include all of the limitations of claim 34, it is respectfully submitted that these claims are also allowable for at least the reasons stated above. Accordingly, Appellants respectfully submit that the rejections of claims 35-46 under 35 U.S.C. § 112 should be reversed.

For the reasons previously discussed with reference to claim 34, it is respectfully submitted that claim 47 also should not have been rejected under 35 U.S.C. § 112. Because claims 48-51 depend from, and, therefore, include all of the limitations of claim 47, it is respectfully submitted that these claims are also allowable for at least the reasons stated above. Accordingly, Appellants respectfully submit that the rejections of claims 47-51 under 35 U.S.C. § 112 should be reversed.

II. Claims covering software and a “computer system” do not fail to meet the standards of 35 U.S.C. § 101, per se.

Claims 34-51 stand rejected under 35 U.S.C. § 101 for including “software per se that fails to meet 35 USC 101”. (See 04/28/2009 Final Office Action, p. 6, lines 5-6). Once again, the Examiner fails to provide any basis or reference to an authority for such a rejection, other than simply citing the first sentence of 35 U.S.C. § 101. The Examiner reiterates this unsupported assertion in his Response to Arguments in stating: “[t]he claim is interpreted as software; software per se fails to meet the statutory claim.” (See *Id.*, p. 3, lines 9-10). Apparently, the Examiner has made this rejection solely on the basis that these claims are

directed to nothing other than “software per se.” However, in contrast to the Examiner’s line of reasoning, the Federal Circuit has recently addressed this “software per se” argument in In re Bilski, ___ F.3d ___, 2008 WL 4757110, 88 USPQ2d (BNA) 1385 (Fed. Cir. Oct. 30, 2008). (hereinafter “In re Bilski”). In footnote 23, the Federal Circuit states, “although invited to do so by several amici, we decline to adopt a broad exclusion over software or any other such category of subject matter beyond the exclusion of claims drawn to fundamental principles set forth by the Supreme Court.” In re Bilski goes on to state:

Read in context, section 101 gives further reasons for interpretation without innovation. Specifically, section 101 itself distinguishes patent eligibility from the conditions of patentability – providing generously for patent eligibility, but noting that patentability requires substantially more. The language sweeps in “any new and useful process... [and] any improvement.” 35 U.S.C. § 101 (emphasis supplied). As an expansive modifier, “any” embraces the broad and ordinary meanings of the term “process,” for instance. *The language of section 101 conveys no implication that the Act extends patent protection to some subcategories of processes but not others. It does not mean “some” or even “most,” but all.*

Unlike the laws of other nations that include broad exclusions to eligible subject matter, such as European restrictions on *software and other method patents*, see European Patent Convention of 1973, Art. 52(2)(c) and (3), and prohibitions against patents deemed contrary to the public morality, see id. at Art. 53(a), *U.S. law and policy have embraced advances without regard to their subject matter*. That promise of protection, in turn, fuels the research that, at least for now, makes this nation the world’s innovation leader. (See In re Bilski). (Emphasis added).

Accordingly, this unsupported categorical exclusion of software per se imposed by the Examiner appears to be in direct opposition to the holdings by the Federal Circuit. Furthermore, as noted above with reference to the 35 U.S.C. § 112 rejections, one skilled in the art of computer science would understand that the methods performed by the computer system of claim 34 are clearly tied to a structural element of a computer. Thus, Appellants respectfully maintain the position that the language in claim 34 should not have been rejected under 35 U.S.C. § 101. Because claims 35-46 depend from, and, therefore, include all of the limitations of claim 34, it is respectfully submitted that these claims are also allowable for at least the reasons stated above. Accordingly, Appellants respectfully submit that the rejections of claims 35-46 under 35 U.S.C. § 101 should be reversed.

For the reasons previously discussed with reference to claim 34, it is respectfully submitted that claim 47 also should not have been rejected under 35 U.S.C. § 101. Because claims 48-51 depend from, and, therefore, include all of the limitations of claim 47, it is respectfully submitted that these claims are also allowable for at least the reasons stated above. Accordingly, Appellants respectfully submit that the rejections of claims 47-51 under 35 U.S.C. § 101 should be reversed.

- III. Claims 1, 2, 4-7, and 10-68 should not be rejected under 35 U.S.C. § 103(a) as being unpatentable over U.S. Patent No. 5,136,709 to Shirakabe in view of U.S. Patent No. 5,778,226 to Adams.

Claims 1, 2, 4-7 and 10-68 stand rejected under 35 U.S.C. § 103(a) as being unpatentable over U.S. Patent No. 5,136,709 to Shirakabe et al. (hereinafter “Shirakabe”) in view of U.S. Patent No. 5,778,226 to Adams et al. (hereinafter “Adams”). (See 04/28/2009 Final Office Action, p. 6, lines 20-21).

Initially, it should be noted that in the Response to Arguments, the Examiner, provides an uncited and unsupported statement, and thus once again fails to provide any authority on which to base his argument. Specifically, the Examiner provides an unsupported definition of Dynamic Link Libraries (DLL). (See Id., p. 2, lines 9-30). Due to the fact that the Examiner fails to offer any citation or authority for this definition, Appellants shall interpret this definition to be based on the Examiner’s personal knowledge. However, as noted in Appellants’ 06/29/2009 Response, and as noted in the M.P.E.P., “there must be some form of evidence in the record to support an assertion of common knowledge. See Lee, 277 F.3d at 1344-45, 61 USPQ2d at 1434-35 (Fed. Cir. 2002); Zurko, 258 F.3d at 1386, 59 USPQ2d at 1697 (holding that general conclusions concerning what is ‘basic knowledge’ or ‘common sense’ to one of ordinary skill in the art without specific factual findings and some concrete evidence in the record to support these findings will not support an obviousness rejection).” (See MPEP § 2144.03(b)). The Examiner does not provide any evidence that his definition of a DLL would be basic knowledge to one of ordinary skill in the art. The only support that the Examiner provides for his assertion

is his personal knowledge in stating, “[i]t should be noted that”...” Furthermore, according to the M.P.E.P., “if the examiner is relying on personal knowledge to support the finding of what is known in the art, the examiner must provide an affidavit or declaration setting forth specific factual statements and explanation to support the finding.” (See MPEP § 2144.03(c); and 37 CFR 1.104(d)(2)). Thus, as argued in the 06/29/2009 Response, Appellants respectfully submit that without the requisite affidavit or declaration to support the Examiner’s use of personal knowledge is improper. Accordingly, in the 06/29/2009 Response, Appellants had requested for the Examiner to produce authority for his definition of DLLs. This request for the production of any authority or support from the Examiner was not addressed in the 08/05/2009 Advisory Action. Furthermore, as noted in the 06/29/2009 Response, any opinion held by the Examiner as to whether the claimed language conforms to or “redefines” his personal definition of a DLL should not serve as a basis to reject a claim.

Referring now to the arguments in which the Examiner provided support for, claim 1 states, *inter alia*, “[a] system for dynamically linking application code created by a programmer *into a running operating system kernel*, comprising: an environment library comprising one or more routines for *insulating the application code from the operating system environment* and for implementing a uniform execution environment, wherein the insulating of the application code allows the application code to perform a function by executing a routine that is standard to the application code instead of by using a routine that pertains to the running operating system kernel... the execution library comprises one or more routines for transparently *loading the loadable module into the running operating system kernel...*” (Emphasis added).

Claims 1 differ from Shirakabe for at least the following reasons. First, as will be discussed in detail below, each of the independent claims performs a step of “insulating” the application code from the operating system kernel space. Second, it is important to note that all of the steps performed in each of the claims is performed on a “*running operating system kernel*.” It appears that this is not the case for the Shirakabe reference, and this will be discussed in greater detail below.

It should be noted that support and definitions for the term “insulating” are found in ¶¶ [0035] and [0036] of the Specification. Specifically, paragraph ¶ [0036] describes an initialization routine performs the functions of “insulating” application code from the operating system environment. As noted in the Specification, the insulation functions may include: copying in arguments, creating kernel/user channels connecting the loadable module to the executable program, requesting a block of memory from the operating system and storing the application code as a “task,” putting the data describing the application code on a “task list.” It is respectfully submitted that this insulating process is a distinguishing factor over the Shirakabe reference. According to the claims in the present invention and an example provided in the Specification, a driver developer can insulate a driver from non-relevant operating system internal changes. (See Specification, ¶ [0050]). However, the systems and processes disclosed in the Shirakabe reference fail to teach or suggest such precautionary measures. While the Shirakabe reference describes a method “capable of incrementally adding a driver to an operating system by use of the same ordinary software as that used in the static link method,” Shirakabe is silent on “insulating the application code from the operating system environment,” as recited in claim 1. (See Shirakabe, col. 3, lines 47-53). In reference to writing data from the driver into an area of the kernel, the Shirakabe reference states a first address, “XXX,” indicates an address in a main memory in which the operating system is loaded at an execution. (See Id., col. 5, lines 53-58). According to the Shirakabe reference, that a PUTDATA procedure is used to access an area of the kernel and write data in the area. (See Id., col. 7, lines 19-24). Therefore, by using an access address, such as XXX, the data of a work area WKi is read with a GETDATA procedure and data is written with the PUTDATA procedure. (See Id., col. 7, lines 19-24). While this operation provides an access from the driver to an area in the kernel of a load module, Shirakabe does not teach or describe the insulating process recited in claim 1. Specifically, Shirakabe does not teach or describe, “storing data of the application code *in a task list within a block of memory of the kernel space* managed by the initialization module.”

It should be noted that, for the sake of argument, the Examiner may consider the driver definition table of Fig. 8 of the Shirakabe reference to be analogous to the “task list” recited in claim 1. However, as depicted in Fig. 8 of the Shirakabe reference, the driver definition table does not reside within the kernel, and thus is external to the memory of the kernel

space. Accordingly, the driver definition table is not equivalent to the task table recited in the claims.

Finally, it should be noted that the Shirakabe reference differs from the claims described above in that each claim recites, “loading the loadable module into the *running* operating system kernel.” It appears that this is not the case for the Shirakabe reference. According to Shirakabe, “[t]he load module of the generate operating system is loaded, when the computer system is initiated, in the memory 91 so as to be executed by the processor 90.” (See Id., col. 10, lines 17-20). Therefore, Shirakabe fails to teach or suggest loading a module into a running kernel. In further support of this argument, Appellants direct the Examiner to following excerpt from the Shirakabe reference:

First, description will be given of the reason why the address solution is required to be achieved *when the setup or initiation is achieved on the computer system*. In the setup operation of the computer system, the operating system is loaded in the main memory so as to set the system to a state capable of coping with interruptions from input/output devices. Consequently, in a case where the address solution has not been effected at the setup of the computer system, since the routine to operate in association with interruptions from the input/output devices is in kernel of the load module 4, when the routine calls the driver 6, a wrong address is accessed as a result, which may lead to a program runaway in some cases. *In order to avoid this disadvantage, the address solution is conducted at the setup of the computer system*. In the setup operation of the computer system, *the reset state is first established and then the loading of the operating system* and the initialization of the respective tables in the operating system are effected in an interrupt disabled state; in consequence, as a portion of the processing above, there is operated the driver table initializing routine 24, which will be described in the following paragraphs. (See Id., col. 8, lines 4-26). (Emphasis added).

Accordingly, in order to avoid “program runaway,” Shirakabe requires for the computer system to establish a “reset state” prior to loading the load module on the operating system. This teaches away from the recitations of the above claim, which state that the loading of the module is performed while the operating system kernel is running. Therefore, in contrast to the Examiner’s assertions, Shirakabe fails to teach or suggestion, “the execution library comprises one or more routines for loading the loadable module *into the running operating system kernel...*”

Finally, it should be noted that claim 1 recites “wherein the insulating of the application code allows the application code to perform a function by executing a routine that is standard to the application code instead of by using a routine that pertains to the running operating system kernel.” The advantage of this feature is summarized at paragraph [0037], which states:

As discussed above, environment library 112 includes one or more routines for insulating application code 102 from the operating system environment and implementing a uniform execution environment. That is, instead of depending on a changeable and specialized operating system interface, the program can use standard “C” interfaces--or alternatively, any standard application programming interface. For example, in the current embodiment of the invention, instead of using a Linux kernel “sys_open” operation applied to some terminal device so that the program can output data, the program can simply use the standard “printf” routine. Instead of using some OS dependent routine for generating a thread, the program can use “pthread_create”--a POSIX standard function. (See Specification, p. 3, ¶ [0037]).

In contrast to claim 1, the Shirakabe reference requires an application code to use a non-standard routine that is specific to the operating system environment of the kernel to which it is linked. Specifically, linkage editor 7 links kernel 3 and driver 5, and as a result of this linkage, driver 5 calls for the execution of a routine in the kernel 3. Specifically, the Shirakabe reference describes a kernel routine KSUBm. The linkage editor 7 creates a load module 26 corresponding to the driver 5 and a load module 4 corresponding to the kernel. The purpose is so that “by use of the load module 26 containing the driver 6 thus separately generated, the routine in the load module 4 of the kernel 3 can be called.” (See Shirakabe, col. 6, lines 27-30). Thus, no insulation of the kind recited in the claim occurs in the Shirakabe reference; indeed, quite the opposite occurs since the driver is required by the linkage editor 7 to call the routine of the kernel. As stated in the Background of the present specification, this absence of insulation from the operating system environment is disadvantageous because it requires the application code to depend on OS dependent routines, which is more complex, time-consuming, and error-prone than merely using the routines standard to the application code. Furthermore, neither the Examiner’s Response to Arguments nor the Claim Rejections, pinpoints any language within the Shirakabe disclosure that teaches or suggests each of the limitations recited in claim 1, such as, at

least, "...an environment library comprising one or more routines for insulating the application code from the operating system environment and for implementing a uniform execution environment, *wherein the insulating of the application code allows the application code to perform a function by executing a routine that is standard to the application code instead of by using a routine that pertains to the running operating system kernel...*"

As for Adams, it shows the same type of non-insulating linkage between driver and kernel that was shown to be deficient in the Shirakabe reference. Specifically, as seen in Fig. 1 of the Adams reference, interface 4 provides a linkage between device driver description tables 8 and kernel 1 that does not insulate driver from the operating system environment in the manner recited in the claim. (See Adams, Fig. 1). Accordingly, withdrawal of the rejection of claim 1 is respectfully requested. Because claims 2, 4, and 10-20 depend from, and, therefore, include all of the limitations of claim 1, it is respectfully submitted that these claims are also allowable for at least the reasons stated above. Accordingly, Appellants respectfully submit that the rejections of claims 1, 2, 4, and 10-20 under 35 U.S.C. § 103, as being unpatentable over Shirakabe in view of Adams, should be reversed.

For at least the reasons previously discussed with reference to claim 1, it is respectfully submitted that claim 5 is also allowable over Shirakabe in view of Adams. Because claims 6 and 7 depend from, and, therefore, include all of the limitations of claim 5, it is respectfully submitted that these claims are also allowable for at least the reasons stated above. Accordingly, Appellants respectfully submit that the rejections of claims 5-7 under 35 U.S.C. § 103, as being unpatentable over Shirakabe in view of Adams, should be reversed.

For at least the reasons previously discussed with reference to claim 1, it is respectfully submitted that claim 21 is also allowable over Shirakabe in view of Adams. Because claims 22-33 depend from, and, therefore, include all of the limitations of claim 21, it is respectfully submitted that these claims are also allowable for at least the reasons stated above. Accordingly, Appellants respectfully submit that the rejections of claims 21-33 under 35 U.S.C. § 103, as being unpatentable over Shirakabe in view of Adams, should be reversed.

For at least the reasons previously discussed with reference to claim 1, it is respectfully submitted that claim 34 is also allowable over Shirakabe in view of Adams. Because claims 35-43 depend from, and, therefore, include all of the limitations of claim 34, it is respectfully submitted that these claims are also allowable for at least the reasons stated above. Accordingly, Appellants respectfully submit that the rejections of claims 34-43 under 35 U.S.C. § 103, as being unpatentable over Shirakabe in view of Adams, should be reversed.

For at least the reasons previously discussed with reference to claim 1, it is respectfully submitted that claim 47 is also allowable over Shirakabe in view of Adams. Because claims 48-51 depend from, and, therefore, include all of the limitations of claim 47, it is respectfully submitted that these claims are also allowable for at least the reasons stated above. Accordingly, Appellants respectfully submit that the rejections of claims 47-51 under 35 U.S.C. § 103, as being unpatentable over Shirakabe in view of Adams, should be reversed.


For at least the reasons previously discussed with reference to claim 1, it is respectfully submitted that claim 52 is also allowable over Shirakabe in view of Adams. Because claims 53-68 depend from, and, therefore, include all of the limitations of claim 52, it is respectfully submitted that these claims are also allowable for at least the reasons stated above. Accordingly, Appellants respectfully submit that the rejections of claims 52-68 under 35 U.S.C. § 103, as being unpatentable over Shirakabe in view of Adams, should be reversed.

8. Conclusion

For the reasons set forth above, Appellants respectfully request that the Board reverse the above-discussed rejections of claims 1, 2, 4-7, and 10-68.

Respectfully submitted,

Date: October 13, 2009

By: 
Michael J. Marcin (Reg. No. 48,198)

Fay Kaplun & Marcin, LLP
150 Broadway, Suite 702
New York, NY 10038
Tel.: (212) 619-6000
Fax: (212) 619-0276

CLAIMS APPENDIX

1. (Previously presented) A system for dynamically linking application code created by a programmer into a running operating system kernel, comprising:

an environment library comprising one or more routines for insulating the application code from the operating system environment and for implementing a uniform execution environment, wherein the insulating of the application code allows the application code to perform a function by executing a routine that is standard to the application code instead of by using a routine that pertains to the running operating system kernel; and

a build system for constructing a loadable module from the application code and the environment library and for constructing a standard executable program from the loadable module and an execution library, wherein

the loadable module including a module input and a module output, and wherein

the execution library comprises one or more routines for transparently loading the loadable module into the running operating system kernel, passing arguments to the loadable module, and terminating and unloading the loadable module after receiving a termination signal, the one or more routines of the execution library setting up input/output channels by connecting a standard input and a standard output of the running operating system kernel to the module input and the module output.

2. (Original) The system of claim 1, further comprising an infrastructure library comprising one or more routines executed prior to loading the loadable module into the running operating system kernel and/or after unloading the loadable module from the kernel.

3. (Canceled)

4. (Previously presented) The system of claim 1, wherein the standard executable program may be in several files or a single file.

5. (Previously presented) A method, comprising:

- creating a loadable module, the loadable module including a module input and a module output;

- creating an executable program; and

- executing the executable program, wherein the executable program performs a method comprising the steps of:

 - setting up input/output channels by connecting a standard input and a standard output of a running operating system kernel to the module input and the module output;

 - inserting the loadable module into address space of the running operating system kernel, wherein, once the loadable the module is inserted into the address space, the loadable module begins to execute; and

 - waiting for the loadable module to connect via kernel/user channels and then connecting those kernel/user channels to the input/output channels, wherein the executable program is insulated from the running operating system kernel, and wherein the insulating of the executable program allows the executable program to perform a function by executing a routine that is standard to the executable program instead of by using a routine that pertains to the running operating system kernel.

6. (Previously Presented) The method of claim 5, wherein after the loadable module is inserted into the address space the loadable module performs a method comprising the steps of:

- creating kernel/user channels;

- creating a thread to execute application code; and

- waiting for the thread to complete.

7. (Original) The method of claim 6, wherein the method performed by the loadable module further includes the step of freeing resources after the thread completes.

8. (Canceled)

9. (Canceled)

10. (Previously presented) The system of claim 1, wherein one of the one or more routines of the execution library includes code for executing a utility for installing the loadable module into the running operating system kernel.

11. (Previously presented) The system of claim 10, wherein the utility for installing the loadable module into the running operating system kernel is the insmod program.

12. (Previously presented) The system of claim 1, wherein the build system includes instructions for compiling the application code into object code.

13. (Previously presented) The system of claim 12, wherein the build system further includes instructions for linking said object code with object code from the environment library to produce a linked object module.

14. (Previously presented) The system of claim 13, wherein the build system further includes instructions for converting the linked object module into a C code array.

15. (Previously presented) The system of claim 13, wherein the build system further includes instructions for compiling the C code array to produce an object file and for linking said object file with object code from the execution library to produce the standard executable program.

16. (Previously presented) The system of claim 1, wherein the environment library includes one or more routines to create kernel/user channels.

17. (Previously presented) The system of claim 1, wherein the environment library includes one or more routines to create a thread to execute the application code.

18. (Previously presented) The system of claim 17, wherein the environment library includes one or more routines for freeing resources and unloading the loadable module when the thread completes.

19. (Previously presented) The system of claim 1, wherein the environment library includes one or more routines for (a) copying in arguments; (b) creating communication channels that connect the loadable module to the executable program; (c) requesting a block of memory from the operating system and storing a structure therein that describes the application code; and (d) putting data describing the application code on a task list.

20. (Previously presented) The system of claim 19, wherein the environment library further includes one or more routines for (a) removing said data describing the application code from the task list; (b) closing said communication channels that connect the loadable module to the executable program; and (c) freeing the block of memory that was requested from the operating system.

21. (Previously presented) A computer readable storage medium including a set of instructions executable by a processor, the set of instructions operable to:

- a first set of computer instructions for insulating application code from an operating system environment, wherein the insulating of the application code allows the application code to perform a function by executing a routine that is standard to the application code instead of by using a routine that pertains to the running operating system kernel;

- a second set of computer instructions for constructing a loadable module from the application code and the first set of computer instructions, the loadable module including a module input and a module output; and

- a third set of computer instructions for constructing an executable program from the loadable module and a fourth set of computer instructions; wherein

- the fourth set of computer instructions includes computer instructions for transparently loading the loadable module into a running operating system kernel, passing arguments to the loadable module, and terminating and unloading the loadable module from the running operating system kernel after receiving a termination signal, the fourth set further includes at least one routine setting up input/output channels by connecting a standard input and a standard output of the running operating system kernel to the module input and the module output.

22. (Previously presented) The computer readable medium of claim 21, wherein the computer instructions for loading the loadable module into the running operating system kernel include computer instructions for executing a utility for installing the loadable module into the running operating system kernel.

23. (Previously presented) The computer readable medium of claim 22, wherein the utility for installing the loadable module into the running operating system kernel is the insmod program.

24. (Previously presented) The computer readable medium of claim 21, wherein the second set of computer instructions includes instructions for compiling the application code into object code.

25. (Previously presented) The computer readable medium of claim 24, wherein the second set of computer instructions further includes instructions for linking said object code with object code from the environment library to produce a linked object module.

26. (Previously presented) The computer readable medium of claim 25, wherein the third set of computer instructions includes instructions for converting the linked object module into a C code array.

27. (Previously presented) The computer readable medium of claim 26, wherein the third set of computer instructions further includes instructions for compiling the C code array to produce an object file and for linking said object file with object code from a library to produce the executable program.

28. (Previously presented) The computer readable medium of claim 21, wherein the first set of computer instructions includes instructions for creating kernel/user channels.

29. (Previously presented) The computer readable medium of claim 21, wherein the first set of computer instructions includes instructions for creating a thread to execute the application code.

30. (Previously presented) The computer readable medium of claim 29, wherein the first set of computer instructions includes instructions for freeing resources and unloading the loadable module when the thread completes.

31. (Previously presented) The computer readable medium of claim 21, wherein the first set of computer instructions includes instructions for (a) creating communication channels that connect the loadable module to the executable program; (b) requesting a block of memory from the operating system and storing a structure therein that describes the application code; and (c) putting data describing the application code on a task list.

32. (Previously presented) The computer readable medium of claim 31, wherein the first set of computer instructions further includes instructions for (a) removing said data describing the application code from the task list; (b) closing said communication channels that connect the loadable module to the executable program; and (c) freeing the block of memory that was requested from the operating system.

33. (Previously presented) The computer readable medium of claim 21, wherein the executable program may be in several files or a single file.

34. (Previously presented) A computer system, comprising:

first means for insulating application code from an operating system environment, wherein the insulating of the application code allows the application code to perform a function by executing a routine that is standard to the application code instead of by using a routine that pertains to the running operating system kernel;

second means for constructing a loadable module from the application code and the first means, the loadable module including a module input and a module output;

third means for constructing an executable program from the loadable module; and

fourth means for transparently loading the loadable module into a running operating system kernel, passing arguments to the loadable module, and terminating and unloading the loadable module from the running operating system kernel after receiving a termination signal, the fourth means includes at least one routine setting up input/output channels by connecting a

standard input and a standard output of the running operating system kernel to the module input and the module output.

35. (Previously presented) The computer system of claim 34, wherein means for loading the loadable module into the running operating system kernel include means for executing a utility for installing the loadable module into the running operating system kernel.

36. (Previously presented) The computer system of claim 35, wherein the utility for installing the loadable module into the running operating system kernel is the insmod program.

37. (Previously presented) The computer system of claim 34, wherein the second means includes means for compiling the application code into object code.

38. (Previously presented) The computer system of claim 37, wherein the second means further includes means for linking said object code with object code from the environment library to produce a linked object module.

39. (Previously presented) The computer system of claim 38, wherein the third means includes means for converting the linked object module into a C code array.

40. (Previously presented) The computer system of claim 39, wherein the third means further includes instructions for compiling the C code array to produce an object file and for linking said object file with object code from a library to produce the executable program.

41. (Previously presented) The computer system of claim 34, wherein the first means includes means for creating kernel/user channels.

42. (Previously presented) The computer system of claim 34, wherein the first means includes means for creating a thread to execute the application code.

43. (Previously presented) The computer system of claim 42, wherein the first means includes means for freeing resources and unloading the loadable module when the thread completes.

44. (Previously presented) The computer system of claim 34, wherein the first means includes means for (a) creating communication channels that connect the loadable module to the executable program; (b) requesting a block of memory from the operating system and storing a structure therein that describes the application code; and (c) putting data describing the application code on a task list.

45. (Previously presented) The computer system of claim 44, wherein the first means further includes means for (a) removing said data describing the application code from the task list; (b) closing said communication channels that connect the loadable module to the executable program; and (c) freeing the block of memory that was requested from the operating system.

46. (Previously presented) The computer system of claim 34, wherein the executable program may be in several files or a single file.

47. (Previously presented) A computer system for dynamically linking application code created by a programmer into a running operating system kernel, comprising:

- means for creating a loadable module, the loadable module including a module input and a module output; and

- means for creating an executable program that is configured to perform a method comprising the steps of:

- setting up input/output channels by connecting a standard input and a standard output of the running operating system kernel to the module input and the module output;

- inserting the loadable module into address space of the running operating system kernel, wherein, once the loadable module is inserted into the address space, the loadable module begins to execute; and

- waiting for the loadable module to connect via kernel/user channels and then connecting those kernel/user channels to the input/output channels, wherein the executable program is insulated from the running operating system kernel, and wherein the insulating of the executable

program allows the executable program to perform a function by executing a routine that is standard to the executable program instead of by using a routine that pertains to the running operating system kernel.

48. (Previously presented) The computer system of claim 47, wherein the loadable module is configured to perform a method after the loadable module is inserted into the operating system address space, wherein said method comprises the steps of:

- creating kernel/user channels;
- creating a thread to execute the application code; and
- waiting for the thread to complete.

49. (Previously presented) The computer system of claim 48, wherein the method performed by the loadable module further includes the step of freeing resources after the thread completes.

50. (Previously presented) The computer system of claim 47, wherein the step of inserting the loadable module into an operating system address space includes the step of creating a child process, wherein the child process replaces its image with the insmod process image.

51. (Previously presented) The computer system of claim 50, wherein the step of inserting the loadable module into an operating system address space further includes the step of piping the loadable module to the insmod process.

52. (Previously presented) A method for dynamically linking application code created by a user into a running operating system kernel, comprising:

- constructing a loadable module from application source code written by a user, the loadable module including a module input and a module output;
- setting up input/output channels by connecting a standard input and a standard output of the running operating system kernel to the module input and the module output;
- creating an executable program, wherein the executable program is configured to transparently load the loadable module into the running operating system kernel;

executing the executable program, thereby loading the loadable module into the running operating system kernel; and

unloading the loadable module from the running operating system kernel by sending a termination signal to the executable program, wherein the executable program is insulated from the running operating system kernel, and wherein the insulating of the executable program allows the executable program to perform a function by executing a routine that is standard to the executable program instead of by using a routine that pertains to the running operating system kernel.

53. (Previously presented) The method of claim 52, wherein the application source code is an ordinary application program.

54. (Previously presented) The method of claim 52, wherein the step of constructing the loadable module from the application source code consists essentially of executing a pre-defined makefile.

55. (Previously presented) The method of claim 52, further comprising the step of providing a makefile to the user, wherein the user performs the step of constructing the loadable module by executing the makefile after the user has created the application code.

56. (Previously presented) The method of claim 52, further comprising the step of providing the user with a library comprising object code, wherein the step of constructing the loadable module from the application source code comprises the steps of compiling the application source code into object code; linking the object code with object code from the library to produce a linked object module; and converting the linked object module into a C code array.

57. (Previously presented) The method of claim 56, wherein the step of constructing the loadable module further comprises the step of compiling the C code array to produce an object file.

58. (Previously presented) The method of claim 57, further comprising the step of providing the user with a second library comprising object code, wherein the step of constructing the

executable program comprises the steps of linking the object file with object code from the second library.

59. (Previously presented) The method of claim 56, wherein the library includes one or more routines to create kernel/user channels.

60. (Previously presented) The method of claim 56, wherein the library includes one or more routines to create a thread to execute the application code.

61. (Previously presented) The method of claim 60, wherein the library includes one or more routines for freeing resources and unloading the loadable module when the thread completes.

62. (Previously presented) The method of claim 56, wherein the library includes one or more routines for (a) copying in arguments; (b) creating communication channels that connect the loadable module to the executable program; (c) requesting a block of memory from the operating system and storing a structure therein that describes the application code; and (d) putting data describing the application code on a task list.

63. (Previously presented) The method of claim 62, wherein the environment library further includes one or more routines for (a) removing said data describing the application code from the task list; (b) closing said communication channels that connect the loadable module to the executable program; and (c) freeing the block of memory that was requested from the operating system.

64. (Previously presented) The method of claim 52, wherein the executable program is configured to set up input/output channels.

65. (Previously presented) The method of claim 52, wherein the executable program is configured to execute a utility for installing the loadable module into the running operating system kernel.

66. (Previously presented) The method of claim 65, wherein the utility for installing the loadable module into the running operating system kernel is the insmod program.

67. (Previously presented) The method of claim 65, wherein the step of inserting the loadable module into an operating system address space includes the step of creating a child process, wherein the child process replaces its image with the insmod process image.

68. (Previously presented) The method of claim 67, wherein the step of inserting the loadable module into an operating system address space further includes the step of piping the loadable module to the insmod process.

EVIDENCE APPENDIX

No evidence has been submitted herewith or is relied upon in the present appeal.

RELATED PROCEEDINGS APPENDIX

There are no related proceedings or decisions that relate to the present appeal.